# 一种倒排索引压缩方法 *

白福均 [a]，高建瓴 [a]，李宛蓉 [b]，贺思云 [a]，肖绍武 [a]

(贵州大学 a. 大数据与信息工程学院; b. 档案馆, 贵阳 550025)

**摘　要：**高效地访问倒排索引是搜索引擎快速响应用户查询的关键，而压缩倒排列表是提高搜索引擎性能的最重要的手段之一。针对自适应分段压缩 ASCS 算法进行了研究，对于 ASCS 算法中采用的均匀分段方式并非最优分段问题，提出以人工蜂群算法优化 ASCS 算法中的分段方式；对于 ASCS 算法考虑序列占用空间的影响因素过于单一问题，提出多因素下的改进算法；对于分布不均的长序列在 ASCS 算法下压缩率不理想问题，提出先排序后差分编码操作后再以 ASCS 算法压缩。通过对比实验证明优化改进后的算法可以较显著的压缩倒排索引。

**关键词：**搜索引擎；倒排索引；索引压缩；人工蜂群算法；ASCS 算法

**中图分类号：**TP391　　doi: 10.3969/j.issn.1001-3695.2017.06.0647

# Method of inverted index compression

Bai Fujun[a], Gao Jianling[a], Li Wanrong[b], He Siyun[a], Xiao Shaowu[a]

(*a. College of Big Data & Information Engineering*, *b. Archives Guizhou University*, *Guiyang* 550025, *China*)

**Abstract:** Efficient access to the inverted index is a key aspect for a search engine to achieve fast response times to users' queries. While compression of its posting lists is one of the most important methods to improve the performance of search engine. Segmentation method optimized by ABC algorithm in ASCS algorithm was proposed for the problem of ASCS algorithm that it adopts uniform segmentation instead of optimal segmentation; The ASCS algorithm only considers an influencing factor and ignores the influence of other factors ; The ratio of compressing long sequence of uneven distribution is unsatisfactory with ASCS algorithm, it was adopted that process integer sequence with sorting and differential encoding before ASCS algorithm. Simulation experiments show that the improved algorithm has significantly increased compression ratio of inverted index file comparing with ASCS algorithm.

**Key Words:** search engine; inverted index; index compression; artificial bee colony; ASCS algorithm



## 0　引言

随着互联网特别是移动互联网的迅猛发展，导致网络信息量越来越庞大，这给信息的存储和检索带来了很大的挑战。如何在海量的信息集合中高效的定位、查找所需的目标信息是迫切需要解决的问题。

倒排索引[1]是现代搜索引擎的索引模型，对于每个索引词汇(term)都对应着一个倒排列表(posting list)，倒排列表包含了词汇所在的文档 ID(DocID)，以及在文档中的频率(frequencies)和位置(positions)等信息。倒排索引文件就是一系列 term 和 posting list 的集合，其结构如表 1 所示。压缩倒排索引即是压缩 term 和 posting list。为了更好的存储和压缩倒排列表，通常是将各个词汇的 DocID、frequencis 和 positions 分开存储。这样，压缩倒排列表就是压缩一系列由 DocID、frequencies、positions 组成的整数序列。对于每个 DocID，其在磁盘中占用的存储空间取决于文档集中最大文档编号的大小，这显然会浪费巨大的空间资源。

表 1　倒排索引结构

| Term | Posting list<DocID,freq,(pos1,...)> |
|---|---|
| term1 | <d1,2,(p2,p7)>, <d2,1,(p6)>,⋯ |
| term2 | <d2,1,(p4)>,⋯ |
| term3 | <d1,1,(p1)>,<d2,2,(p3,p9)>,<d4,2,(p3,p12)> |
| ... | ... |

倒排索引压缩技术目前应用很广泛，也有很多算法或者编码在搜索引擎中被采用。一种普遍的做法是通过存储倒排列表

**基金项目：**贵州省档案局科研项目（2015D001）；贵州省科学技术基金项目（黔科合 J 字［2015］2045）；贵州大学研究生创新基金资助项目（研理工 2017014，研理工 2017016）

**作者简介：**白福均（1990-），男，贵州习水人，硕士研究生，主要研究方向为数据挖掘、云计算（fjbai901124@126.com）；高建瓴（1969-），女，福建南安人，副教授，硕导，主要研究方向为数据挖掘、云计算；李宛蓉（1969-），女，贵州贵阳人，本科，主要研究方向为档案管理；贺思云（1993-），女，贵州瓮安人，硕士研究生，主要研究方向为聚类、半监督聚类；肖绍武（1992-），男，湖北武人，硕士研究生，主要研究方向为食品安全、舆情分析.

中 DocID 和 positions 的差分值[2]（d-gaps）而不是存储他们的实际值来减小倒排列表占用的存储空间。对于目前的压缩算法而言可以通过编码性质[3]将它们分为按位压缩(bitwise)、字节对齐压缩(byte aligned)、字对齐压缩(word aligned)、自压缩(oblivious)、自适应列表压缩(list-adaptive)。对于自压缩，它仅通过自身的值来进行编码，只考虑自身信息而忽略了相邻信息和全局信息。

Unary[4]是将一个正整数 $n$ 编码为 $(n-1)$ 个位 1 和 1 个位 0（例如：5 被编码为 11110），这种编码方式适合小整数的编码，对于大整数编码反而会占用更多的存储空间；Elias Gamma[5]编码会将整数 $n$ 编码成两个部分：长度部分和数据部分。长度部分为 $n$ 的二进制长度的 Unary 编码，而数据部分为 $n$ 的二进制表示法去掉最高位，即 $n=U(|B(n)|)+B(n)^{'}$，比如 6 被编码为 110|10。同样地，Gamma 编码不适合编码大整数；Elias Delta[6]编码与 Gamma 一样将整数 $n$ 编码成长度部分和数据部分，只不过长度部分表示成 $n$ 的二进制长度的 Elias Gamma 编码，数据部分和 Elias Gamma 编码的数据部分完全一致，都为 $n$ 的二进制表示法去掉最高位即 $n=EG(|B(n)|)+B(n)^{'}$，例如：6 被编码成 1101|10，此编码对大整数的编码效率较高；Golomb[7]编码是将整数 $n$ 被除数 $d$ 整除得到商 $q$ 和余数 $r$，对于压缩序列来说，通常取除数 $d=0.69*avg$，$avg$ 是序列的平均数，整数 $n$ 的 Golomb 编码为 $n=Unary(q)+B(r)$；Rice[8]编码是 Golomb 编码的升级版，这里将除数 $q$ 指定为 2 的指数幂，这样做的好处是使得除法运算转换成高效的移位运算，编码速率更快；Extended Golomb[9]编码是将整数 $n$ 反复被除数 $d$ 整除 $k$ 次，直到商 q=0 为止，即 $n=Unary(k)+Code(r_k,r_{k-1},\ldots,r_1)$。后续又陆续出现了 Fast Extended Golomb[10]编码(FEGC)、Re-Orded FEGC[11]编码(RFEGC)、Block RFEGC[11]编码(BRFEGC)、Re-Ordered Fast Modified Extended Golomb[12]编码(RFMEGC)、Run-Length Encoding RFEGC[12]编码(RLETFEGC)、动态规划 FREGC[12]编码(DPRFEGC)等 Extended Golomb 编码(EGC)的改进编码；PackedBinary[13]编码是一种定位编码方式，它每次编码一个序列的整数，首先选出序列中的最大整数的有效位宽 $w$，然后将序列中的所有整数都用 $w$ 位进行编码；Variable Byte[14]是一种字节对齐编码方式，它用每个字节的低 7 位表示整数 $n$ 二进制部分，最高位用 0 或者 1 表示整数 $n$ 是否被编码完成，1 表示编码完成，0 表示编码未完成，例如 201 被编码为 10000001 01001001；Simple[15]家族编码用于压缩整数序列，编码的思想是在一个字里尽可能地压缩多个整数，例如可以在 1 个字的前 4 位存放描述后 28 位中各个整数所占的位数，对于 $\{509,510,511\}$ 可编码为 0100|111111101|111111110|111111111；Frame of reference[16] (FOR)编码一次编码若干个整数（例如 128 个），计算这 128 个整数的最大值 $M$ 与最小值 $m$ 的差值，然后以二进制形式存储 $m$，其他整数以到最小值 $m$ 的差值的二进制形式进行存储，每个整数所占的位数为 $b=\lceil\log_2(M-m+1)\rceil$；Patched frame of reference[17]编码(PFOR)是为了解决 FOR 编码在存在较大异常值 $M$ 时使得 $b$ 值过大而使压缩率变差而提出来的一种改进编码算法，PFOR 编码为大多数整数（例如 90%）选择参数 $b$，而不能用 $b$ 位表示的大整数被称为异常值（值大于 $2^b$），因此 PFOR 编码使用 FOR 正确编码正常值，对于异常值所占的 $b$ 位用来存储下一个异常值的位置，真实的异常值不压缩地存储在 FOR 编码输出之后；当 $b$ 位无法表示异常值的间隔时将增加 $b$ 的值，从而使得压缩率下降，为了改进和增强 PFOR 编码，NewPFD[18]编码、OptPFD[18]编码和 FastPFOR[19]被陆续的提出使得 PFOR 编码日趋完善。

为了提高倒排索引的压缩率，本文对自适应分段压缩 ASCS 算法[20]进行深入研究。ASCS 算法存在以下缺陷：a)ASCS 算法中采用的均匀分段方式并非最优分段导致压缩率不佳；b)ASCS 算法考虑影响序列占用空间的因素过于单一而无法有效压缩数据；c)分布不均的长整数序列在 ASCS 算法下压缩率不理想。针对 ASCS 算法存在以上不足，提出以下改进方法：以人工蜂群算法优化 ASCS 算法中的分段方式；提出一种在多因素下的综合改进方法；对整数序列进行排序差分预处理后再压缩。实验表明改进优化后的算法大大提高了倒排索引的压缩率。

## 1 相关算法介绍

### 1.1 CSN 编码

对于正整数序列 $x_1,x_2,x_3,\ldots,x_n$，可以用一个哥德尔数[21]唯一表示，哥德尔数是基于质数因式分解的编码系统，其公式如下：

$$enc(x_1,x_2,x_3,\ldots,x_n)=2^{x_1}\cdot 3^{x_2}\cdot 5^{x_3}\cdot\ldots\cdot p_n^{x_n} \tag{1}$$

其中 $p_n$ 是连续递增的质数。从式(1)可以看出哥德尔数编码会随着整数序列值和长度的增加而指数增长，不适用于长整数序列压缩。基于哥德尔数的思想引出 CSN-1 数[20]编码的定义：

**定义 1** 对于一个长度为 $n$，存在最小元素为 1 的整数序列 $x_1,x_2,x_3,\ldots,x_n$，取 $T=\max(x_1,x_2,x_3,\ldots,x_n)+1$，由递推式(2)可得 CSN 数编码。

$$\begin{cases} a_0=0 \\ a_k=T\cdot a_{k-1}+x_k, 1\le k\le n \\ CSN=a_n \end{cases} \tag{2}$$

这样，就可以将一个整数序列压缩为一个 CSN 数编码。但在 CSN-1 中，对于 $x_i\le 0$ 或者最小元素不为 1 时，此编码方式将不再适用。因此 CSN-2 在 CSN-1 的基础上进行了改进提升，CSN-2 数[20]定义如下：

**定义 2** 对于一个长度为 $n$ 的整数序列 $x_1,x_2,x_3,\ldots,x_n$，取 $m=\min(x_1,x_2,x_3,\ldots,x_n)$，$T=\max(x_1,x_2,x_3,\ldots,x_n)-m+2$，由递推式(3)可得 CSN 数编码。

$$\begin{cases} a_0=0 \\ a_k=T\cdot a_{k-1}+(x_k-(m-1)), 1\le k\le n \\ CSN=a_n \end{cases} \tag{3}$$

这样，就提升了 CSN 编码的适应度。

对于 CSN 编码的解码，实际上是 CSN 编码的逆过程,其递推式(4)如下：

$$\begin{cases} a_0 = CSN \\ x_k = rem(\frac{a_{k-1}}{T}) + (m-1), k \ge 1 \\ a_k = \left\lfloor \frac{a_{k-1}}{T} \right\rfloor, a_k > 0 \end{cases} \tag{4}$$

其中：$rem(*)$ 为取余运算，$\lfloor * \rfloor$ 为取整（取商）运算。当 $a_k = 0$ 时，递推结束，将序列 $x$ 倒序即可还原得到原编码序列。

### 1.2 自适应分段 ASCS 算法

要解压一个 CSN 编码需要两个参数，即 $T$ 和 $m$，因此压缩一个长度为 $n$ 的整数序列时需要存储 $CSN$、$T$ 和 $m$ 三个参数，对于 CSN 数编码的通用式(5)计算如下：

$$\begin{aligned} CSN &= a_n \\ &= T \cdot a_{n-1} + (x_n - (m-1)) \\ &= T \cdot (T \cdot a_{n-2} + (x_{n-1} - (m-1))) \\ &\quad + (x_n - (m-1)) \\ &= T^2 \cdot a_{n-2} + T \cdot x_{n-1} + x_n \\ &\quad - (T+1)(m-1) \\ &= T^2 \cdot (T \cdot a_{n-3} + (x_{n-2} - (m-1))) \\ &\quad + T \cdot x_{n-1} + x_n - (T+1)(m-1) \\ &= T^3 \cdot a_{n-3} + T^2 \cdot x_{n-2} + T \cdot x_{n-1} + x_n \\ &\quad - (T^2 + T + 1)(m-1) \\ &= \cdots \\ &= T^n \cdot a_0 + T^{n-1} \cdot x_1 + T^{n-2} \cdot x_2 + \cdots \\ &\quad + T \cdot x_{n-1} + x_n + (T^{n-1} + T^{n-2} + \cdots \\ &\quad + T + 1)(m-1) \\ &= T^{n-1} \cdot x_1 + T^{n-2} \cdot x_2 + \cdots + T \cdot x_{n-1} \\ &\quad + x_n + (T^{n-1} + T^{n-2} + \cdots + T + 1)(m-1) \\ &= T^{n-1} \cdot (x_1 + m - 1) + T^{n-2} \cdot (x_2 + m - 1) \\ &\quad + \cdots + T \cdot (x_{n-1} + m - 1) + x_n + m - 1 \\ &= \sum_{i=0}^{n-1} T^i \cdot (x_{n-i} + m - 1) \end{aligned} \tag{5}$$

因此存储一个序列整数需要的空间为

$$\begin{aligned} & S(x_1, x_2, x_3, \cdots, x_n) \\ &= \log_2 CSN + 2\sigma \\ &= \log_2 \sum_{i=0}^{n-1} T^i \cdot (x_{n-i} + m - 1) + 2\sigma \end{aligned} \tag{6}$$

其中 $\sigma$ 为存储一个整数所占用的空间。

从式(6)可以看出，影响一个整数序列所占的空间大小的因素除了与序列本身相关的取值和长度外，还与序列值的分布和最大最小值有关，参数 $T$ 与空间 $S$ 构成幂函数关系，而参数 $m$ 与空间 $S$ 构成线性函数关系。一个取值较为集中的序列可以得到较好的压缩效果。

当 $T$ 值过大时，会使得序列的压缩效果不理想。自适应分段 ASCS[20]算法通过对长序列进行分段压缩，使每段的 $T$ 值和序列长度相对于长序列来说都有所减小，实现了各段更好的压缩，最终使得长序列的 $T$ 平均期望值达到最小，从而实现更佳的压缩效果。

自适应分段 ASCS 算法描述如下：

a)初始化分段数 b=1,最优分段数 p=1（即不分段情况下），计算 $T$ 值；

b)计算各分段序列的 $T$ 值，并计算长序列的平均 $T$ 值为 $t$，若 $b < n-1$ 则继续运算，否则运算结束；

c)若 $t < T$，则 $T = t$， $p = b$；

d)增加一个分段，即 $b = b+1$，回到 Step2 继续迭代运算。

通过上述算法步骤后，可以求得使长序列平均 $T$ 值最小的分段数，进而得到较好的压缩效果，分段压缩后所占的空间为各段所占空间之和,即

$$\begin{aligned} & S(x_1, x_2, x_3, \cdots, x_n) \\ &= \sum_{j=1}^{p} \log_2 \sum_{i=0}^{n_j-1} T_j^i \cdot (x_{j,n_j-i} + m_j - 1) + 2\sigma p \end{aligned} \tag{7}$$

其中:$p$ 为分段数，$T_j^i$ 为第 $j$ 分段参数 $T$ 的 $i$ 次幂，$n_j$ 为第 $j$ 分段所包括的整数个数，$x_{j,n_j-i}$ 为第 $j$ 分段第 $(n_j - i)$ 个元素的值，$m_j$ 为第 $j$ 个分段的参数 $m$（最小值）。

## 2 ASCS 算法的优化与改进

虽然分段压缩 ASCS 算法可以减小各小序列 $T$ 值和序列长度，但是却有可能增加各个分段的最小值 $m$，使得长整数序列压缩效果不甚理想。从式(7)可以看出，通过减小各分段的参数 $T$、$m$ 以及序列值 $x_i$ 可以有效的改善 ASCS 算法的压缩率。本文结合倒排索引的特点对分段压缩 ASCS 算法进行如下改进：

### 2.1 对序列重排序并差分编码

在倒排索引中，DocID 通常是按顺序递增排列且不重复地出现在倒排列表中。因此结合倒排列表的特点，首先对待压缩序列按递增顺序进行重新排列，然后将序列中的后项与前项作差进行初次编码。通过以上两步操作可以有效的减小序列的值，在同样的分段方式下可以减小各分段的参数 m 和序列值 $x_i$，显然，相比于原序列，新序列的编码效果会更好。例如，对于序列 $X = (2,4,5,8,12,13,20,22,25)$，差分编码后变为 $X^{'} = (2,2,1,3,4,1,7,2,3)$，可以看出差分编码后序列 $X^{'}$ 相比于 $X$ 有明显的减小，若对原序列进行如下分段 $<3,6,9>$，即 $X = (2,4,5|8,12,13|20,22,25)$，在同样分段方式下差分编码为 $X^{'} = (2,2,1|3,4,1|7,2,3)$，此时，对于原序列 $X$ 有 $(m_1 = 2, T_1 = 5), (m_2 = 8, T_2 = 7), (m_3 = 20, T_3 = 7)$，而对差分序列 $X^{'}$ 有 $(m_1^{'} = 1, T_1^{'} = 3), (m_2^{'} = 1, T_2^{'} = 4), (m_3^{'} = 2, T_3^{'} = 6)$，比较新旧序列可以看出，新序列不仅显著的减小的序列的值，而且在同分段方式下，各段的参数 $m$，$T$ 均有所改善，因此本改进显得十分必要。

### 2.2 人工蜂群算法优化序列分段方式

在分段压缩 ASCS 算法中，对于确定的分段数，ASCS 算法会采用均匀的分段方式对长序列进行切分，但均匀分段并不一定是最优的分段方式。本文采用群智能优化算法人工蜂群

[22](ABC)算法优化 ASCS 算法在确定分段数情况下的分段方式，以得到最优分段方式。人工蜂群算法优化的过程如下：

a)初始化 $NP$ 个蜜源 $X$，$X=(X_1,X_2,X_3,\cdots,X_{NP})$，每个蜜源对应一个可行解（即一种分段方式，表示为分段坐标），其中每个蜜源都是 $d$ 维向量 $X_i=(X_{i1},X_{i2},X_{i3},\cdots,X_{id})$（即蜜源编码），这里的 $d=p-1$（即分段数减 1），蜜源初始化公式如式(8)所示。

$$\begin{cases} X_{ij}=randInt[1,n-1] \\ X_{ik}=randInt[1,n-1] \end{cases} \tag{8}$$

其中：$j,k\in(1,2,\cdots,d)$，$j\neq k$，$i\in(1,NP)$，且 $X_{ij}\neq X_{ik}$，$n$ 为序列长度，即蜜源的每维均为 1 到 $(n-1)$ 之间的随机整数且互不相等。

b)引领蜂计算对应蜜源的适应度值，并搜索新蜜源。适应度计算公式如下：

$$fit_i=\frac{1}{\overline{T_i}}=\frac{b}{\sum_{l=1}^{b}T_{il}} \tag{9}$$

其中: $fit_i$ 为第 $i$ 个蜜源的适应度值，$\overline{T_i}$ 为第 $i$ 个蜜源对应分段方式下的平均 $T$ 值，$T_{il}$ 为第 $i$ 个蜜源对应分段方式下第 $l$ 段的 $T$ 值，$b$ 为当前分段数。引领蜂搜索新蜜源公式如下：

$$V_{ij}=X_{ij}+randInt[-a,a] \tag{10}$$

其中: $i\in(1,NP)$，$j$ 为 $(1,d)$ 间的随机整数，$a$ 为范围控制参数，即引领蜂在其附近范围内搜索新蜜源，如果找到比当前蜜源更为优质的蜜源(适应度值更高)，则当前蜜源位置移至新蜜源处，否则蜜源不移动，蜜源未更新计数加 1。

c)跟随蜂采用轮盘赌的方式以式(11)概率性地选择蜜源进行采蜜，更新蜜源，更新方式与引领蜂同样为式(10)，同样地，若蜜源未更新，则未更新计算加 1。跟随蜂选择蜜源概率计算公式如下：

$$P_i=\frac{fit_i}{\sum_{i=1}^{NP}fit_i} \tag{11}$$

d)当某一解(蜜源)的未更新次数大于或等于阈值参数 limit 时，引领蜂变为侦测蜂跳出局部最优解，根据式(8)重新获得一个新蜜源，并置对应的蜜源未更新计数为 0。

e)保存当前全局最优解信息，并判断算法终止条件，若迭代次数达到预设参数 maxCycle，则终止算法并输出最佳分段方式结果，否则回到 b)不断循环迭代更新蜜源，直到算法达到终止条件为止。

### 2.3 对 ASCS 算法的改进

ASCS 算法着力于找到一个使得待压缩长序列的平均 $T$ 值最小的分段方式进行分段压缩，根据式(6)可以看出虽然参数 $T$ 对序列的压缩性能有着最直接的影响，但 ASCS 算法并没有从全局角度去考虑问题，忽略掉了同样对压缩性能有影响的参数 $m$，$x$，$p$，因此，对 ASCS 算法的压缩率还有提高的空间。

结合本文提出的用人工蜂群算法优化 ASCS 分段方式，本文提出一种改进方法，即在对待压缩长序列进行分段时，不再将长序列的平均 $T$ 值作为目标函数，而是将影响序列占用空间的所有因素考虑进去，以式(7)作为目标函数。这样，蜜源的适应度计算式(9)更改为式(12)。

$$fit_i=\frac{1}{\sum_{j=1}^{p}\log_2\sum_{i=0}^{n_j-1}T_j^i\cdot(x_{j,n_j-i}+m_j-1)+2\sigma p} \tag{12}$$

在每次 ASCS 算法迭代中，分段数 $p$ 为定值，各个蜜源所对应的各个分段的参数 $T$、$m$ 以及分段序列值之均为定值，因此可以根据式(12)计算出各蜜源的适应度，最终实现对长整数序列的高效压缩。

### 2.4 改进优化的 ASCS 算法程序流程图

通过上述改进优化的 ASCS 分段压缩算法伪代码描述如下：

输入：待压缩整数序列

输出：CSN 序列，以及相关参数 $T$、$m$

流程图如图 1 所示。

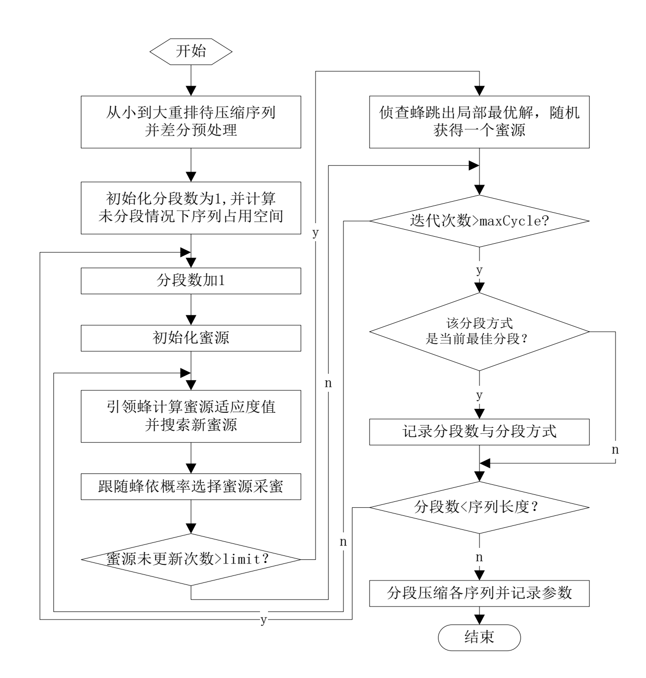



图 1 改进优化后的程序流程图

## 3 实验与结果分析

本实验综合考虑整数序列以整数形式存储(Int)、整数转字符串形式存储(String)、CSN 编码存储、差分处理后的 CSN 编码存储(dCSN)、CSCA 编码存储、差分 CSCA 编码存储(dCSCA)、人工蜂群算法优化后的 CSCA 编码存储(CSCAABC)、差分后的 CSCAABC 编码存储(dCSCAABC)以及在综合三者（差分、人工蜂群算法优化和多因素综合考虑改进）优化改进后存储(dCSCAABCP)在不同规模下的序列压缩结果进行比较。序列的压缩率定义如下：

$$rate=\frac{S_{前}-S_{后}}{S_{前}}\times100\% \tag{13}$$

S 为序列占用空间位数。

本实验选择长度不一的 9 个整数序列作为实验数据，其中

蜜源寻优范围控制参数 $a$ 设置为 2，蜜源未更新参数 limit 设置为 5，算法迭代终止参数 maxCycle 设置为 250，验证比较各个序列在不同编码上的压缩率。数据详细描述如下表 2 所示，各个序列在不同编码上的的占用空间数和相对于整数存储的压缩率分别如表 3 和图 1 所示。

表 2　实验数据描述

| 序列 | 长度 | 描述 |
|---|---|---|
| s1 | 10 | 0 到 20 之间的不重复随机整数 |
| s2 | 20 | 0 到 50 之间的不重复随机整数 |
| s3 | 50 | 0 到 100 之间的不重复随机整数 |
| s4 | 100 | 0 到 300 之间的不重复随机整数 |
| s5 | 200 | 0 到 500 之间的不重复随机整数 |
| s6 | 500 | 0 到 1000 之间的不重复随机整数 |
| s7 | 1000 | 0 到 2000 之间的不重复随机整数 |
| s8 | 5000 | 0 到 10000 之间的不重复随机整数 |
| s9 | 10000 | 0 到 20000 之间的不重复随机整数 |

表 3　序列在不同编码上的空间(bit)占用情况

| | s1 | s2 | s3 | s4 | s5 | s6 | s7 | s8 | s9 |
|---|---|---|---|---|---|---|---|---|---|
| String | 704 | 1792 | 4672 | 11744 | 24128 | 62080 | 141920 | 783104 | 1741920 |
| Int | 320 | 640 | 1600 | 3200 | 6400 | 16000 | 32000 | 160000 | 320000 |
| CSN | 104 | 195 | 774 | 1101 | 2487 | 5038 | 17418 | 66480 | 142912 |
| dCSN | 94 | 182 | 740 | 958 | 2273 | 4123 | 16563 | 52064 | 104928 |
| CSCA | 96 | 188 | 750 | 976 | 2362 | 4338 | 16378 | 57984 | 131968 |
| dCSCA | 91 | 176 | 700 | 922 | 2173 | 4019 | 15418 | 49936 | 115968 |
| CSCAABC | 86 | 172 | 716 | 947 | 2184 | 3581 | 15818 | 51504 | 119584 |
| dCSCAABC | 76 | 161 | 701 | 802 | 1971 | 3365 | 15603 | 48448 | 116864 |
| dCSCAABCP | 69 | 143 | 646 | 712 | 1765 | 3096 | 15139 | 40832 | 98368 |

从表 3 可以看出，同一整数序列在不同的编码方式上占用的存储空间差异较大。在未压缩情况下，以字符串形式存储的序列较以整数形式存储的序列占用的存储空间大得多，因此在实际中大多数未压缩序列以整数形式存储。

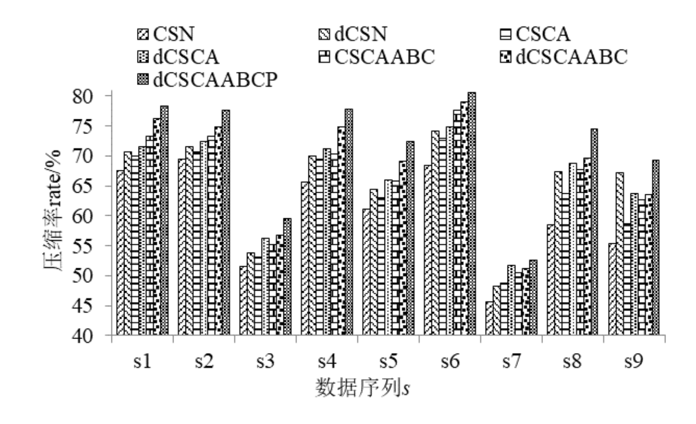



图 2　序列在不同编码上的压缩率对比

从图 2 来看，不同的序列在同一编码上表现出不同的压缩率，这与序列自身相关；而从同一序列来看，作为 CSN 编码改进的 dCSN 和 CSCA 编码较 CSN 编码均有一定的压缩率提升；同样，作为 CSCA 编码改进的 dCSCA 和 CSCAABC 编码较 CSCA 编码也有压缩率的提升；最后综合改进的 dCSCAABCP 编码在不同的序列上均具有最好的压缩率。

## 4　结束语

本文通过排序差分初次编码对待压缩数据预处理，以降低序列值以及提升序列值的聚集度；通过人工蜂群算法(ABC)优化 CSCA 压缩算法的分段方式进一步提升序列压缩率；通过将式(6)作为 CSCA 分段的依据，从整体上压缩序列。实验对比发现，改进后的 dCSCAP 算法对整数序列具有较好的压缩率。虽然本改进算法具有较好的压缩率，但由于 dCSCAP 算法会在编码过程中不断迭代寻优，因此在一些需要实时索引和检索的场合不太适用，但对于一些定期更新的索引检索引擎来说，该算法具有较高的适用性。此外，对于载海量数量情况下，可以考虑采用分布式计算框架多机并行编码来提高编码速度，以增强算法的适用性。在解码方面，改进后的 dCSCAP 算法较 CSCA 算法来说仅仅增加了差分步骤，因此在解码时只需较 CSCA 算法多做一次减法运算即可，该步骤对算法的解码性能影响微乎其微。综上所述，本文提出的改进算法具有一定的现实意义。

## 参考文献：

[1] Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze. An Introduction to Information Retreival [M]. Cambridge: Cambridge University Press, 2009: 67-68.

[2] Hersh W. Managing gigabytes—compressing and indexing documents and images (2nd Edition) [J]. Information Retrieval, 2001, 4 (1): 79-80.

[3] Ounis I, Amati G, Plachouras V, et al. Terrier: a high performance and scalable information retrieval platform [C]// Proc of the Osir Workshop. 2006.

[4] Elias P. Universal codeword sets and representations of the integers [M]. [S. l. ] : IEEE Press, 1975.

[5] Manber U, Myers G. Suffix arrays: a new method for on-line string searches [J]. Siam Journal on Computing, 1990, 22 (5): 319-327.

[6] Golomb S W. Run-length encodings [J]. IEEE Trans on Information Theory, 1966, 12 (3): 399-401.

[7] Rice R, Plaunt J. Adaptive variable-length coding for efficient compression of spacecraft television data [J]. IEEE Trans on Communication Technology, 2003, 19 (6): 889-897.

[8] Somasundaram K, Domnic S. Extended GOLOMB code for integer representation [J]. IEEE Trans on Multimedia, 2007, 9 (2): 239-246.

[9] Domnic S, Glory V. Inverted file compression using EGC and FEGC [J]. Procedia Technology, 2012, 6 (4): 493-500.

[10] Domnic S, Glory V. Re-ordered FEGC and block based FEGC for inverted file compression [J]. International Journal of Information Retrieval Research, 2013, 3 (1): 71-88.

[11] 毛福林. 倒排索引压缩算法研究 [D]. 北京: 北京交通大学, 2015.

[12] 闫宏飞, 张旭东, 单栋栋, 等. 基于指令级并行的倒排索引压缩算法 [J]. 计算机研究与发展, 2015, 52 (5): 995-1004.

[13] Williams H E, Zobel J. Compressing integers for fast file access [J]. Computer Journal, 1999, 42 (3): 193-201.

[14] Anh V N, Moffat A. Inverted index compression using word-aligned binary codes [J]. Information Retrieval, 2005, 8 (1): 151-166.

[15] Goldstein J, Ramakrishnan R, Shaft U. Compressing relations and indexes [C]// Proc of International Conference on Data Engineering, 1998: 370-379.

[16] Zukowski M, Heman S, Nes N, et al. Super-scalar RAM-CPU cache compression [C]// Proc of International Conference on Data Engineering. Washington DC: IEEE Computer Society, 2006: 59.

[17] Yan H, Ding S, Suel T. Inverted index compression and query processing with optimized document ordering [C]// Proc of International Conference on World Wide Web. 2009: 401-410.

[18] Lemire D, Boytsov L. Decoding billions of integers per second through vectorization [M]. Hoboken: Wiley, 2012.

[19] 特日跟，江晟，李雄飞，等. 基于整数数据的文档压缩编码方案 [J]. 吉林大学学报, 2016, 46 (1): 228-234.

[20] Gödel K. Uber Formal Unentscheidbare Sätze der Principia Mathematica und Verwandter System, I [J]. Monatshefte Für Mathematik, 1931, 38 (1): 173-198.

[21] 秦全德，程适，李丽，等. 人工蜂群算法研究综述 [J]. 智能系统学报, 2014, 9 (2): 127-135.